Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-2437 **March 2001**

Strategies and Experiences Using High Performance Fortran

Dale Shires, Ram Mohan, and Andrew Mark
Corporate Information and Computing Directorate, ARL

Abstract

Since its beginnings in the 1950's, the Fortran language has been the language of choice for most scientific and engineering programming. Compilers, seizing on the simplicity of the language, have historically generated highly-optimized machine code. High performance Fortran (HPF) is a relative new addition to the Fortran dialect. It is an attempt to provide an efficient high-level Fortran parallel programming language for the latest generation of parallel computers. Its success has been debatable. By operating at a high level, the HPF standard does not provide some low-level detail required to achieve maximum performance in a multiprocessor system. Message passing using highly-tuned libraries, such as the message passing interface (MPI), will more often than not require less wall clock time than a comparable HPF code. However, the HPF language and its compilers continue to mature and improve. HPF provides a convenient way to represent parallelism for those most comfortable with data parallel methodologies. As such, it can require a shorter time to solution and provide an acceptable level of efficiency. This report discusses our experiences with the language, as well as coding strategies and vendor-specific "hooks" that can be used to boost performance.

Acknowledgments

This research was made possible by a grant of computer time and resources by the Department of Defense High Performance Computing Modernization Program. Additional support was provided by the Integrated Modeling and Testing (IMT) Common High Performance Software Support Initiative (CHSSI).

INTENTIONALLY LEFT BLANK.

Table of Contents

INTENTIONALLY LEFT BLANK.

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

The Fortran programming language has a long history in the scientific and engineering communities. A vast array of Fortran legacy and dusty-deck codes exist, and the simplicity of the language and related compiler-generated code efficiency help to keep it to a large extent the language of choice for parallel program development in scientific computing [1]. The language continues to evolve and mature with compilers now available for the newest Fortran 90 and Fortran 95 standards. These latest versions begin to provide some facets of object-oriented program design and language extension by providing modules and abstract data typing [2].

However, Fortran has continued to lack a set of features needed for portable and efficient programming on parallel architectures. The High Performance Fortran Forum (HPFF) met for the first time in 1992 with a goal of providing a set of parallel extensions to the Fortran 90 language [3]. Fortran 90 and HPF are closely associated with the data parallel programming model. Data parallelism implies a simultaneous operation applied across a large set of data. For example, array syntax notation in Fortran 90 implies than an identical operation be applied to all elements of the array. In most cases, this parallelism is at the statement level in a source program, and is therefore often known as fine-grain parallelism. This was a natural fit for the massively parallel, fine-grain computers popular at the time, such as the Connection Machine CM-2 and Mas Par MP-2.

Several vendors supply HPF compilers for today's high performance computing systems. Data parallel, also known as SIMD (single instruction, multiple data), computers have largely been replaced by coarse-grain computers where processing units can work independently [4]. The data parallel methodology is general enough that it can be translated in practice to work quite well on these multiple instruction, multiple data (MIMD) computers. This report presents our experiences in using HPF for the COMPOSE (Composite Manufacturing Process Simulation Environment) suite of codes under development at the U.S. Army Re-

search Laboratory (ARL) and the University of Minnesota. It describes the state of the HPF compiler and provides some programming strategies. It also gives some performance statistics. A more detailed comparison between performance achieved by the HPF compiler and an explicit message passing approach using MPI is beyond the intended scope of this report, but will be provided in follow-on manuscripts. Indeed, extension to shared memory parallelism using techniques such as OpenMP are also planned for comparison.

2. Algorithm Design for Parallel Computers

2.1 Parallel Architectures. The obvious distinguishing feature of parallel computers is their ability to have numerous processors executing at once. There are, however, several means to this end. Numerous classes of architectures and topologies have been developed during the short history of parallel computing. Symmetric multi-processors (SMPs) provide an easy parallel programming environment and can be inexpensive to assemble. These machines use a shared memory in which all processors have the same access penalty to memory. Memory access can potentially limit scalability due to contention in a bus-based topology. Accordingly, most of these architectures use no more than 64 processors. Tiered memory systems using cache have been used to mitigate these effects. The Sun Microsystems Ultra Enterprise 10000 is an example of an SMP. Sun has reported good scalability in clustering its SMPs [5].

Arrays and clusters have been developed where each node could be an SMP or something as simple as a personal computer [6]. They are most often connected using high-speed networks. They are scalable, but provide no shared memory and can be difficult to program. Massively parallel processors (MPPs) most often feature distributed memories with large numbers of processors. The processor interconnection network topologies have ranged from hypercubes to three-dimensional (3-D) meshes. Examples include the Connection Machine-5 (CM-5) that had the ability to function in multiple instruction-multiple data (MIMD) mode, where each processor can execute different instructions at the same time on different data

sets, or single instruction-multiple data (SIMD) mode, where each processor executed the same instruction in lockstep. The Cray T3E-1200 is also an MPP using a tightly-coupled 3-D bidirectional torus configuration. Lastly, scalable symmetric multiprocessors (S^2MPs) have been designed to provide the best features of SMPs, but they allow for MPP scalability. The SGI Origin 2000 and Origin 3800 are examples of these architectures. They employ a distributed shared memory and large cache architecture built on a bristled hypercube topology. The design intent is to remain scalable by limiting the number of potential "hops" data must make before arriving at the processor requesting the information.

It should be noted that any discussion of interconnection topologies is mostly educational. New cross-platform tools and standardized parallel programming methodologies have greatly reduced the need for a detailed knowledge of the processor interconnection network. It does remain relevant, however, to have a good understanding of the memory systems on multiprocessor computers. For example, the main solution to overcoming the widening gap between memory and CPU performance has been the development of layered caches. Algorithms or coding practices that do not try to remain "cache friendly" are invariably doomed to poor execution on the majority of high performance computers in use today.

2.2 Parallel Algorithms and Programming. The parallel programmer has several tasks. The first is to design or select an appropriate algorithm for parallel execution. Often, the best sequential algorithm, or the most apparent one, is not the algorithm of choice in a parallel environment. Criteria for selection can include any number of metrics, including wall clock execution time, FLOP (Floating Point Operations per Second) rates, throughput, and implementation difficulty. In most cases, an absolute definition of "performance" is ubiquitous and remains a complex issue. For example, spending an inordinate amount of time to boost FLOP rates on an algorithm that will only be executed infrequently at most hardly seems worth the effort.

Once an algorithm is selected or designed within the performance requirements, it must

then be coded, tested, and optimized. Several parallel programming models may be apparent and applicable to the application at hand. Message passing is widely used. Many tasks may be created with data exchange and interaction between the processes being carried out by sending and receiving messages. The MPI has become the predominate tool in this regard. Data parallelism is also widely found in many applications. Data parallelism exploits the fact that the same operation is often performed on each item in a set of data. A data parallel program is a sequence of such operations. Currently, High Performance Fortran (HPF) is the most widely used language to represent data parallelism. Fortran 90 is a data parallel language in its own right, but it is limited to strict array syntax parallelism. A final methodology is shared memory programming, which allows processes to execute concurrently using a common memory space. Examples include OpenMP, SGI loop-level parallelism, and Cray shared memory programming (SHMEM). Most often, these approaches require programmer-inserted directives into the code to establish parallel regions or indicate safe loops for multithreaded execution.

When mapped to the underlying architecture, each of these models has several issues. Chiefly, these are data placement, the number of processors available, the size of the data, and the coordination of the processors. There is, however, no best answer as to which programming paradigm best fits which particular architecture. Data parallel codes can perform well in distributed shared memory machines as well as systolic SIMD computers. When used on distributed memory or distributed shared memory machines, the compiler typically generates a Single Program-Multiple Data (SPMD) program in which each processor executes the same program on a different data set [7]. However, data communication and layout are controlled by the compiler and cannot be easily optimized if required. Furthermore, the maturation rate of data parallel compilers, especially HPF, has been slow due to the complexity of the task. Therefore, they remain, at least somewhat, in a developmental phase.

MPI codes also create SPMD parallelism. These codes require explicit programmer data decomposition and can be very efficient; they are known as the "assembly language" of par-

allel programming because of the low-level details surrounding the code. Directive-based loop-level parallelism can be scalable, but on shared memory machines, tedious and sometimes pejorative coding styles are often required to achieve the desired cache optimization. Furthermore, loop-level parallelism alone may not properly address all numerical formulations.

The choice in parallel methodology can sometimes be difficult. The following sections describe some of our experiences in parallelizing the main solver in the COMPOSE suite. The physical solutions have been based on domains rendered by finite elements. This code has been parallelized using an element-by-element data parallel approach, and by a message passing approach. This discussion is limited to our experiences and strategies garnered from using the HPF compiler for the data parallel approach. A detailed comparison between the two methods will be reported at a later time.

3. HPF Data Parallel Programming

The High Performance Fortran model encompasses both communication and parallelism. It augments Fortran 90, itself a data parallel language that provides constructs to represent concurrent execution, but not domain decomposition. HPF provides additional parallel directives and data placement capabilities. Communication is realized through data distribution, mapping, and alignment. It is the job of the compiler to effectively map and distribute data. Communication is implicit in the code. Parallelism is effected through several mechanisms, including Fortran 90 style array assignments, parallel library routines, the `FORALL` statement, and the `INDEPENDENT` directive [3]. This list is not complete. Extrinsic procedures are available to allow for other programming paradigms or languages. The language continues to evolve through changes to the standard. Unlike MPI, which is realized through calls to a communication library, HPF is a language. To write the most efficient HPF code possible, it is therefore necessary to understand the way an HPF compiler works.

HPF achieves efficient parallelism through a combination of concurrency and locality of data reference. While the two are interrelated, it is possible to discuss them separately. Concurrency assures that all processors are busy, while locality limits the potential amount of communication found in the concurrent statements. For example, consider the parallel statement `A = B * C` (`A`, `B`, and `C` are all conformable vectors), implying `A(1) = B(1) * C(1)`, `A(2) = B(2) * C(2)`, etc. This statement can proceed concurrently with or without communication required between the processors, depending on how the data were distributed.

In the following sections, some of the parallel constructs available in HPF are discussed. Proper strategies to ensure optimized code generation, as well as good and bad coding practices, are also highlighted. Also discussed is the importance of data mapping to achieve locality of reference and avoid potentially costly communication. In this regard, we also discuss various actions that can be taken to mitigate communications when dealing with unstructured grid data. The HPF compiler referenced later in the report refers to the Portland Group HPF (PGHPF) compiler versions 2.4-4 and 3.0 installed at various Major Shared Resource Centers.

3.1 Concurrency and Parallelism in HPF. The `FORALL` statement provides for parallelism by augmenting and merging the Fortran 90 array assignment and `WHERE` statements. The `FORALL` construct is well defined with no nondeterminacy. A multistatement `FORALL` is interpreted as a series of single `FORALL` statements. A set of valid index values is computed based on an optional mask. The right-hand side is then computed for each of these index values. At the same time, any subexpressions in the left-hand side are evaluated and saved. The computed value of the right-hand side is then assigned to the left-hand side. There are no assumptions made on the order of assignments or evaluations. This gives the compiler some freedom in determining an optimal algorithm within the constraints listed. These constructs are most useful as a generalization of Fortran 90 array assignments with more robust array shaping. The `INDEPENDENT` directive can be applied to either do loops or `FORALL` statements. In the case of do loops, this directive tells the compiler that the loop

iterations do not interfere with each other in any way and may be executed in parallel. In other words, there are no loop-carried data dependencies.

3.1.1 Compiler Capabilities and Programming Strategies. The PGHPF compiler acts as a front-end, translating HPF code into architecture target Fortran 90 code. As such, the parallel programmer is free to utilize any of various well-documented programming styles amiable to high-performance microprocessors. Furthermore, compile line options may be passed directly to the native compiler to control various optimizations.

The PGHPF compiler also allows the user to link to runtime libraries and code generation supporting either message passing interfaces (MPI) or shared memory programming (SMP). Linking with MPI causes the code to use MPI to exchange data between processors. SMP allows the compiler to utilize shared memory references found in shared memory and distributed shared memory machines. SMP was used since it typically shows better performance for our targeted shared memory architectures.

Since HPF is a language, it requires a compiler to generate executable code. This starkly contrasts with MPI, which is not a separate language, but rather calls to a library of functions. The compiler for MPI code is a native compiler (usually C or FORTRAN) that sees the calls to MPI simply as calls to functions. The native compiler has no ability to optimize parallelism through message passing library calls. As such, the MPI programmer has sole responsibility for writing the best possible parallel code. The HPF user can also write clear and precise parallel code, but the compiler is more forgiving of poorly written code. It can also restructure code to promote parallelism.

Consider the case of processor synchronization. In parallel programming, synchronization points are costly. Most often, these points are collective barriers at which each process must arrive before they can all continue. These points may also exist at entry and exit points for code that is sequential and must be processed by only one processor. The HPF compiler tries very hard to remove or limit barrier code generation. One way it does this is through

loop fusing. As an example, consider the following two FORALL loops and scalar initialization of the variable a:

```
forall (i=1:nelem)
   ielaxis(i) = iaxis(ielmat(i))
   elangle(i) = angle(ielmat(i))
end forall

a = b * c / d

forall (I=1:nelem)
   elkxx(i) = kxx(ielmat(i))
   elkxy(i) = kxy(ielmat(i))
   elkyy(i) = kyy(ielmat(i))
end forall
```

As long as it maintains the intent of the FORALL construct, the compiler can generate the most efficient code possible to achieve the objective. The compiler can determine that there are no dependencies between the two loops, and that the second FORALL construct contains no references to the scalar computation. In this case, the compiler can simply move the scalar computation before or after the FORALL statements and fuse the two loops without losing correctness or semantics. This allows for the removal of library calls to begin and end local parallel sections. The HPF compiler moved the scalar computation to a point following the second FORALL, fused the separate FORALL loops, and generated the following intermediate code. The code generated for the programmer-fused loop is identical:

```
 do i$i = i$$l, i$$u
    ielaxis(i$i+ielaxis$sd(10)) = iaxis(ielmat(i$i+ielmat$sd(10))+
+iaxis$sd(10))
    elangle(i$i+elangle$sd1(10)) = angle(ielmat(i$i+ielmat$sd(10))+
+angle$sd(10))
    elkxx(i$i+elangle$sd1(10)) = kxx(ielmat(i$i+ielmat$sd(10))+
+kxx$sd(10))
```

```
    elkxy(i$i+elangle$sd1(10)) = kxy(ielmat(i$i+ielmat$sd(10))+
+kxy$sd(10))
    elkyy(i$i+elangle$sd1(10)) = kyy(ielmat(i$i+ielmat$sd(10))+
+kyy$sd(10))
 enddo
```

This example is somewhat contrived; since these FORALL loops are independent, the source code could have equally been written using INDEPENDENT do loops as well. The FORALL constructs are distinctly different from INDEPENDENT do loops though. The INDEPENDENT do loops can have scalar temporaries. Precedence graphs of the two constructs for similar code bodies are very different and hence require different control and data flow analysis. In fact, the same exercise was attempted using INDEPENDENT do loops to see if the compiler would fuse them. For any of several reasons, the current version of the compiler did not fuse the loops.

While it is the job of the HPF compiler to generate efficient code and communication, the programmer can assist with certain coding styles. Synchronization points can be reduced as well. For example, programming in styles appropriate for systolic SIMD machines should be avoided on MIMD parallel computers. The following code fragment produced intermediate code for the Cray T3E-1200 with barriers following each WHERE statement:

```
where (fillfac < 1.0)
   rhs = cvol * fillold - kp + gkflow
   fillfac = rhs / cvol
end where

where (fillold /= 1.0 .and. fillfac > 0.999)
   fillfac = 1.0
end where

where (fillold /= 1.0 .and. fillfac < 0.0)
```

```
   fillfac = 0.0
end where

where (fillold == 1.0)
   fillfac = 1.0
end where
```

These masked array assignments were rewritten using one INDEPENDENT do loop. This new formulation was more efficient in that it contained no internal synchronization points. It is somewhat unrealistic for the programmer to expect developing compilers to be exhaustive in their ability to optimally process every coding possibility. Indeed, compilers for robust high-level languages often require years to become very efficient at code generation. This further highlights the need for parallel programmers to understand the architecture they will utilize and write appropriate code. The INDEPENDENT do loop should actually perform quite well on many different computers.

The HPF compiler is very good at generating efficient code for RISC-based superscalar processors from Fortran 90 array syntax statements. Consider the following code fragment:

```
slocal = s * wssqrtrec
sk = wssqrtrec * temp * 3.14159
```

These statements appear to be well suited for vector processing. On MIMD shared memory machines, the HPF compiler will convert this code to SPMD execution by creating equivalent FORALL statements. As written, it appears that processor synchronization may occur after the assignment to slocal before the assignment to sk begins. In HPF, it may be tempting to rewrite the code using the INDEPENDENT directive. The following code fragment performs the same function as the array syntax version:

```
!hpf$ independent
```

```
do i = 1, n
         slocal(i) = s(i) * wssqrtrec(i)
         sk(i) = wssqrtrec(i) * temp(i) * 3.14159
      enddo
```

In general, however, the HPF compiler is also very good at Fortran 90 array syntax merging, and rewriting the code is unnecessary. HPF can analyze consecutive array syntax statements and merge them if the meaning is not altered. If the arrays are conformable and distributed identically, the compiler merges the two into a parallel region with no synchronization between the individual statements. The resultant code for the two versions is practically the same.

HPF does share in one of Fortran 90's problems with code optimization and the use of array syntax. Fortran 90 provides the ability for programmers to use assumed-shape dummy arrays as a special type of procedure parameter. Array bounds no longer have to be passed as arguments along with the array to a called procedure. These arrays are declared with a type and rank (the number of comma separated entries), but the size (or bounds) is determined at runtime based on the array passed to the procedure. An explicit interface to the procedure ensures the type, order, and rank are coherent between caller and callee. However, traditional compiler optimizations, such as pipelining and the associated loop fusion to boost pipelining potential, can be taxed by this method. The compiler cannot fuse loops whose bounds cannot be determined. Transitive properties and other interprocedural analyses can be used to assist the compiler in determining whether two arrays are of the same extent, but this is a complicated task. In these instances, it is preferable to use the `INDEPENDENT` directive to ensure that the loops are fused.

In the past, many data parallel languages required the extensive use of array syntax to describe parallelism. CM-Fortran codes for the CM-5 relied heavily on array syntax to achieve parallel execution. While it can be argued that this syntax is easier to read, it has

several potential faults. A major drawback in using array syntax notation for parallelism is that many temporary multidimensional arrays are often required. This problem can quickly get out of hand for codes with large data sets. This was a major reason for establishing the INDEPENDENT do loop and NEW clause construct in HPF.

Consider the following array syntax statements:

```
x(:) = w(2,:)*p(3,:) - w(3,:)*p(2,:)
y(:) = w(3,:)*p(1,:) - w(1,:)*p(3,:)
z(:) = w(1,:)*p(2,:) - w(2,:)*p(1,:)
d(4,:) = sqrt(x(:)*x(:) + y(:)*y(:) + z(:)*z(:))
```

This can be rewritten as an independent loop which allows the conversion of x, y, and z to scalars:

```
!hpf$ independent, new(x, y, z)
      do j = 1, n
         x = w(2,i) * p(3,i) - w(3,i) * p(2,i)
         y = w(3,i) * p(1,i) - w(1,i) * p(3,i)
         z = w(1,i) * p(2,i) - w(2,i) * p(1,i)
         d(4,i) = sqrt(x**2 + y**2 + z**2)
      enddo
```

If the extent of the final dimension is n, we can use only three scalars and reduce memory requirements by $3n$. Furthermore, the loop will require much less storage and will be able to take advantage of scalar values. This can have an enormous impact on cache-based architectures.

It may also be the case that further analysis of the code will allow for scalar replacement of w and p if they are used primarily as temporaries. The code was restructured in critical regions to use the INDEPENDENT clause. Also targeted were small functions that used

multidimensional arrays as temporaries. In many of these functions, scalar replacement was performed by unrolling small inner loops. More thorough findings on the memory conserved through these techniques will be reported in the future. Further analysis of the code will undoubtedly reveal more opportunities for wider use of the INDEPENDENT directive and array to scalar conversion.

Strict use of the array syntax coding style may also be detrimental for another reason. Array syntax code is also difficult for the compiler to analyze. Consider the following code fragment:

```
      aelpk(1,:) = fel(1,1,:)*y(lm(1,:)) +
&       fel(1,2,:)*y(lm(2,:)) + fel(1,3,:)*y(lm(3,:))
      aelpk (2,:) = fel(2,1,:)*y(lm(1,:)) +
&       fel(2,2,:)*y(lm(2,:)) + fel(2,3,:)*y(lm(3,:))
      aelpk (3,:) = fel(3,1,:)*y(lm(1,:)) +
&       fel(3,2,:)*y(lm(2,:)) + fel(3,3,:)*y(lm(3,:))
```

The indirect referencing in y results in a gather operation from potentially remote nodes. As previously noted, HPF is good at array syntax merging. The compiler merged these three loops into one locally executed do loop. However, it also generated approximately 120 lines of code to compute the data communication schedules and other overhead. Furthermore, it primarily used more complicated memory referencing and references into vectors throughout the do loops.

Conversely, consider the following equivalent INDEPENDENT do loop:

```
!hpf$ independent, new(t1, t2, t3)
      do j = 1, nelem
         t1 = y(lm(1,j))
         t2 = y(lm(2,j))
         t3 = y(lm(3,j))
```

```
      aelpk(1,j) = fel(1,1,j)*t1+fel(1,2,j)*t2+fel(1,3,j)*t3
      aelpk(2,j) = fel(2,1,j)*t1+fel(2,2,j)*t2+fel(2,3,j)*t3
      aelpk(3,j) = fel(3,1,j)*t1+fel(3,2,j)*t2+fel(3,3,j)*t3
    enddo
```

Here, the programmer assisted the compiler by telling it certain values will be reused. The code required to compute the communication schedules and other overhead was cut by over 50% to approximately 49 lines.

Also, the concurrent do loop contained direct references to the scalars `t1`, `t2`, and `t3`, and generally had less complicated memory referencing. Most current architectures must use at least two layers of cache to overcome the discrepancy between memory access speeds and processor speeds. The opportunity for independent scalar quantities gives the parallel programmer the ability to write more cache-friendly code. This `INDEPENDENT` directive should also extend well to new techniques and capabilities in the Fortran 90 language, such as abstract data types and pointers, and new approaches for cache optimizations [8]. The execution time for a very small problem was reduced from 24 s using the array syntax version to 21 s using the `INDEPENDENT` formulation. Obviously, larger problems should see more pronounced gains.

3.2 Locality. As with any parallel code, the paramount concern rests in limiting to the greatest extent possible the amount of communication that must occur between the processors in the parallel pool. Determining the optimal distribution of data objects operated on by a program is a global optimization problem, and as such is not tractable. Accordingly, HPF provides directives for data mapping (alignment and distribution) to advise the compiler on how best to distribute data elements to the parallel processors. As would be expected, these directives work best at reducing communication in an environment comprised of regular, grid-based data. For instance, with a 2-D or 3-D grid of data, it is relatively straightforward for the compiler to distribute data evenly across the parallel processors. For

"ghost points," those items on the data mapping borders which are shared between processors, it is also feasible for the compiler to vectorize and agglomerate the data that must be communicated between processors, thus reducing the overall time spent in communication. Efficient scheduling in these cases is also possible to hide memory hierarchy latency.

Communication is implicit in HPF as compared to explicit calls found in message passing codes. While this in principle is a factor to make coding in HPF "easier" than traditional message passing languages, it also represents an area that requires special attention if HPF codes are to perform as well as their message passing counterparts. Communication in HPF results from the interplay between the program being executed and the data layout resulting from the distribution directives. An obvious source of communication is found in collective operations, such as summation reduction. These operations obviously require some sort of cross processor communication. Furthermore, with unstructured finite element meshes, there is the distinct possibility that the HPF data mapping directives will not serve to keep the data as processor-local as possible. We now discuss how these potential bottlenecks can be mitigated.

3.2.1 Collective Operations. Preceding data parallel languages recognized and attempted to address the potential poor performance of collective routines. CM-Fortran, used on the CM-5 computer, used the Connection Machine Scientific Software Library (CMSSL) which was created for array syntax notation and data parallel architectures. The CM-Fortran code used the CMSSL collective routines part_scatter_setup, part_scatter, part_gather_setup, and part_gather. These routines were used to perform partitioned scatter and gather operations, respectively. These routines use source arrays, destination arrays, and pointer arrays containing the scatter/gather pattern. Data are then scattered/gathered from the source array to the destination array.

The CMSSL routines part_scatter_setup and part_gather_setup were available to optimize data locality and reduce the associated communication time. During the setup phase, these

routines analyzed the required communication patterns that would be required and reordered the pointer arrays to achieve better data locality.

These collective routines are also possible in HPF. In HPF, there are no default library routines to do gather operations. The CMSSL equivalent of the two calls

```
call part_gather_setup(lm,.true.,fillfac,setup,ier)
call part_gather(elfill,fillfac,.true.,setup)
```

are performed by nested `INDEPENDENT` do loops

```
!hpf$ independent, new(j)
      do i = 1, ndel
!hpf$ independent
         do j = 1, nelem
            elfill(i,j) = fillfac(lm(i,j))
         enddo
      enddo
```

The scatter operation is slightly different. Here are the example CMSSL routines to perform a scatter operation

```
      call part_scatter_setup(lm, .true., wgnode, setup, ier)
      call part_scatter(wgnode, wel, .true., setup)
```

There is a `sum_scatter` HPF library function to perform the reduction

```
wgnode = sum_scatter(wel, wgnode, lm)
```

The full details of the implementation are hidden, but most likely this call computes a communication schedule for data going to and arriving from remote nodes, moves the data,

and then computes the reduction. It is also possible for robust compilers to perform the reductions locally before sending out the data.

These communication operations can be very expensive. A profile of our code revealed that it was communication bound with well over 50% of the execution time being spent in calls to the sum_scatter library routine. Approximately 20% of the time was spent in code segments performing gather operations. The library routine sum_scatter is called repeatedly, thousands of times even for small problems. Since it is a library routine, our assumption was that each time it was called, a schedule was being computed and executed, and any information gathered by the scheduling algorithm was being discarded before the next call. Conversations with the Portland Group confirmed that this was the case.

Obviously, the ability to reuse communication schedules is essential to getting good performance with this code. The Portland Group has already recognized this need. They supplied us with an experimental release 2.4-dev99a of their HPF compiler. The Cray T3E is the only computer currently targeted in this release. This version of the software allows the programmer to store a pointer to the communication schedule determined by the compiler and reuse it. The schedule can be called repeatedly, hence removing the need to recompute the schedule at each call to sum_scatter. While the details of the communication computation are hidden, it is easy to envision a nonoptimal scheduling algorithm taking at least $O(n^2)$ time, with n being the number of elements in a finite element mesh. The call to `alpkm1 = sum_scatter(aelpk, alpkm1, lm)` is replaced with

```
sked = pghpf_comm_sum_scatter_2(tfill,yl,.true.,lm,.true.)
...
call pghpf_comm_execute(sked, alpkm1, aelpk)
...
call pghpf_comm_free(1,sked)
```

The Portland Group reports that some users have experienced a threefold code speedup after switching to reusable schedules. A marked decrease in execution time was noticed with

reusable schedules. The wall clock execution time for a very small problem (1344 nodes, 2560 elements) using four processors dropped from 28.52 s down to 10.23 s. A 16-processor run of an airframe structure (29,171 nodes, 58,187 elements) dropped from 15534.78 s to 6111.37 s, cutting the time by a factor of 2.5.

While gather operations do not contain the arithmetic reduction, they are equally problematic. As mentioned previously, gathers generate a lot of code to compute off-node data locations. In test runs, the gathers started to take longer than the reusable schedule scatter operations. The ability to reuse communication schedules inside of `do independent` loops used for gathers is also under investigation. Syntax to accomplish this has been proposed by the Japan Association for HPF (JAHPF) [9]. Currently, the ability to reuse communication schedules remains a vendor-specific feature, as the HPF 3.0 standard does not address the issue. It will undoubtedly be incorporated into future standards.

3.2.2 Mesh Reconfiguration. The locality of reference greatly impacts the performance of a data parallel program. HPF provides several directives and distributions to map data and promote locality. Of these, the `DISTRIBUTE` and `ALIGN` directives are most common. The `DISTRIBUTE` directive indicates how an array is to be partitioned to the various processors. Array alignment, to make sure that corresponding entries of different arrays are on the same processor, can be specified using the `ALIGN` directive. For example, in the earlier example of `A = B * C`, by aligning B and C with A, we know that A(1), B(1), and C(1) all reside on the same processor. The array dimensions can be distributed as `*`, `BLOCK`, or `CYCLIC`.

For some applications, such as 2-D image processing routines, these distributions map easily and intuitively to the data to promote locality of reference. However, for unstructured finite element mesh-based data, the data sets are usually element and node based. Depending on the quality of the original finite element mesh, a `BLOCK` or `CYCLIC` distribution of the data will require differing amounts of communication. For example, consider the

BLOCK distributions of the node and element arrays in Figure 1. This figure shows how the element distribution across processors can require much communication, depending on how the elements reference nodal-based data.

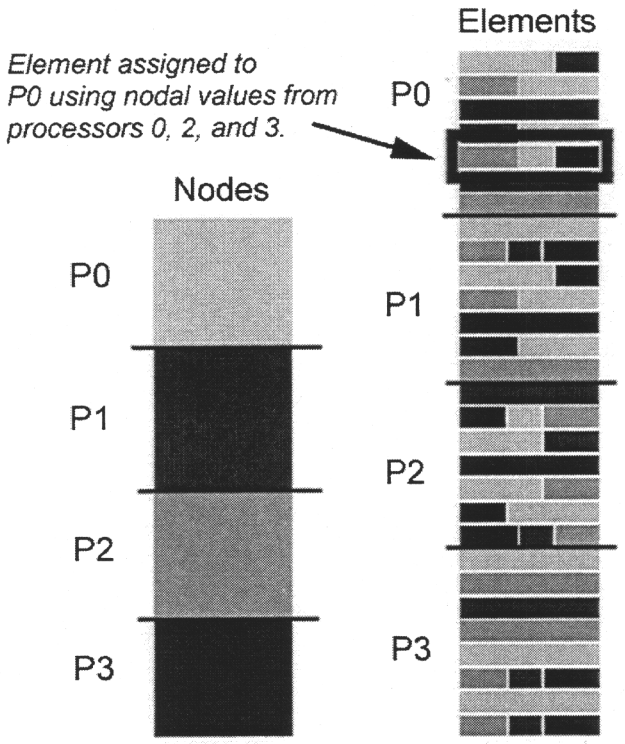

Figure 1: Original Mesh Configuration With High Communication Requirements.

Rarely will even "good" meshes be optimal in all cases. Varying the number of processors alone with the same mesh will cause differing amounts of communication. For example, consider a triangular finite element mesh with an element e consisting of nodes *n1, n2,* and *n3*. Assume there are 10000 elements, 4000 nodes, $e = 1$, $n1 = 1$, $n2 = 2$, and $n3 = 1900$. Further assume that all the data is distributed in block format. If two processors are used, the first 2000 nodes and the first 5000 elements would be local on processor 1. So, there would be no communication required for a computation based on element e. However, if using four processors, node *n3* data would now reside on processor 2 rather than processor 1 and would hence require some communication.

Accordingly, it is extremely advantageous to have a preprocessing step which reorders the data in a smart fashion based on the expected number of processors to be used. This technique is already used in SPMD message passing codes where the input domain is decomposed into a number of partitions equal to the number of processors. The following strategy was employed.

First, the mesh was partitioned using unstructured graph or mesh partitioning software. These packages attempt to divide the mesh, either according the the nodal or element data, into a number of partitions while attempting to limit the number of shared nodes between partitions. This step provided a list of elements for each domain. Second, the domain shared node vectors were computed. Third, for each domain or partition, the nodes that are shared between domains are grouped and renumber first. For example, for partition 0, all the nodes shared with partition 1 are grouped and renumbered, then partition 2, etc. This step helps promote data agglomeration and message vectoring. By placing all of the shared items in a contiguous location, hopefully the compiler will have to do just one send consisting of a memory starting location and vector length. Finally, all of the domain-interior nodes are renumbered. The next domain is then processed in a similar fashion.

This process, if implemented efficiently, can be very fast. This renumbering technique is on the order of $O(n\ log\ n) + O(m)$, where n is the number of nodes that are shared between the various domains, and m is the number of nodes in the mesh. In practice, even though meshes can be quite large, the mesh or graph partitioning step is more onerous and time consuming. Hence, this implementation as described is bounded only by the speed of the mesh partitioning software. Figure 2 shows how nodal renumbering in the elements can reduce the required communications across processor memories. Communications now consist of nodal-based data that must be shared between the domains and artifacts left over from domain partitions not exactly matching the `BLOCK` distribution boundaries as computed by the HPF compiler.

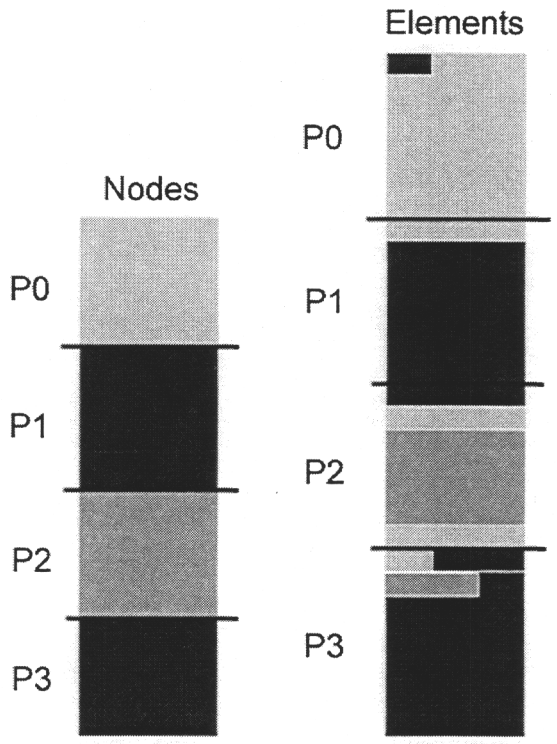

Figure 2: A Better Mesh Configuration Realized by Renumbering.

This extra processing has resulted in reduced execution times in every case. In some cases, the payoff has been outstanding. Using the renumbering approach, the time for the 16-processor simulation mentioned previously (6111.37 s) was cut to 3135.03 s, or roughly in half. We are still assessing the impact of the agglomeration and vectoring steps to see if they are providing any increased efficiency realized by the compiler. At this point it remains doubtful, as even mature compilers are not yet highly skilled at analyzing and restructuring vector access patterns.

Even better execution times are possible with some finite element mesh configurations. Poorly numbered meshes can lead to detrimental performance of the cache hardware on a processor. For example, if a triangular element is comprised of nodes 1, 2, and 8000, any element-based operation will have to gather data from nodal-based arrays in position 1, 2, and 8000. The access to element 1, say in vector `v(1)`, will in effect prefetch the data for

item `v(2)` into cache. However, item `v(8000)` will also have to be loaded and may not be used on the next iteration. The overall performance is hard to determine *a priori* due to the differing cache architectures (set associative, n-way, etc.) and hardware-software interactions (instruction prefetching, speculative code execution, etc.), but it is obvious that this does not lead to cache affinity.

Renumbering techniques were implemented that were proven to improve cache performance by reducing the bandwidth and envelope for sparse matrices [10]. The Reverse Cuthill-McKee (RCM) approach renumbers poorly numbered meshes to reduce the dramatic cardinality changes between connected element nodes. On message passing implementations of COMPOSE, these preprocessing steps have shown roughly a 10% reduction in execution time requirements on the Cray T3E system. The timings reported in this monograph are based on meshes with somewhat poor numbering, and were done prior to implementing RCM. We can roughly assume a similar reduction in the reported timings had the RCM pass been available.

While the aforementioned techniques are very effective at reducing communications, they still do not address the problem of using a mesh that cannot be rigidly broken along domains to fit into the compiler partition sizes. With `BLOCK` distributions, the data is partitioned into contiguous, equal-sized blocks the size of $\frac{N}{P}$, where N is the cardinality of the data set and P is the number of processors. So, even while using an advanced renumbering technique as described previously, some communication will be required to move data for these overlapping points.

One possible way to eliminate these communications for ghost points is through "mesh padding." In this process, a graph partition is determined and renumbered as before. Then, the block distribution boundaries are computed based on the mesh size and compared to how well the graph/finite element partition maps to these block boundaries. Nodes and elements are then created as needed to make the finite element mesh fit the mesh partition. Depending

upon the initial mesh configuration, the number of nodes and elements that have to be created vary from the tens to the hundreds. However, this approach has several drawbacks. The original mesh intent may be lost by adding new nodes and elements. Furthermore, we are adding local computations at the expense of reducing communications.

A better approach has been developed. A new data distribution technique available in the Portland Group HPF compiler release 3.0 addresses these concerns by providing asymmetrical block distribution, thus removing the need for mesh padding [9]. The new approach allows for "soft" `BLOCK` boundaries that can be set by the software or the user at runtime and not the compiler at compile time. Combined with the mesh renumber technique described above, it should limit all communications except for true domain boundary entries. The syntax is

```
!HPF$ DISTRIBUTE A(GEN_BLOCK(DIST)).
```

Here, `A` is the array to be distributed, and `DIST` is an array whose cardinality is the number of processors on which the code will run. Each element of `DIST` contains the size of the data block that is to be placed on the processor. Since the mesh partitioning and renumbering pass has already decomposed the data into domains, it can also provide the `DIST` array with precise numbers as to the size of the asymmetrical blocks. The overall effect should be to limit communication to all but the essential shared nodes and hence reduce execution time. Figure 3 shows the configuration of this optimal data-parallel mesh.

Unfortunately, new languages and compilers do have their faults, as we were unable to utilize this new feature. As released, the compiler is unable to generate an executable code that uses generic block distribution and is compiled with the shared memory addressing mode enabled. In the case of the T3E system, shared memory referencing causes the compiler to generate code that uses very efficient Cray one-sided "get" and "put" communication calls.

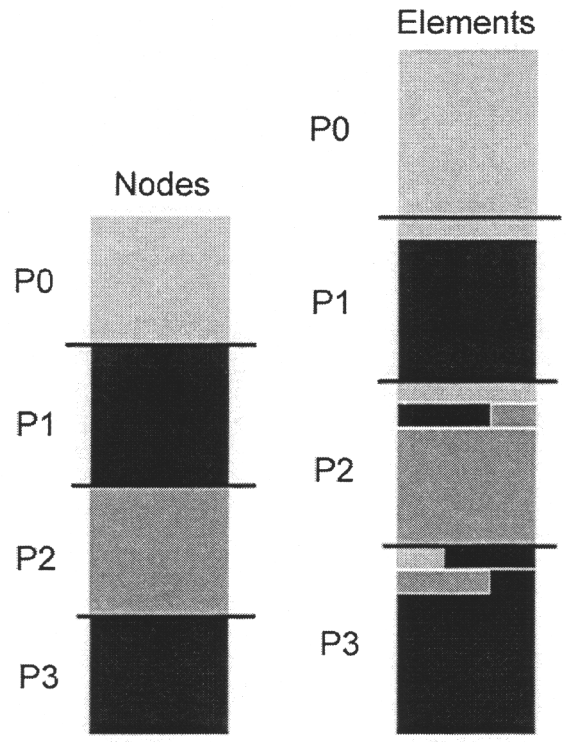

Figure 3: Optimal Mesh Configuration (Renumbering and Soft `BLOCK` Boundaries).

After disabling shared memory compilation to use generic blocks, the compiler began to replicate arrays that were used in gather loops. The arrays used in gather loops were not conformable (not uncommon for element and node arrays); rather than generate potentially costly communication runtime library calls, the compiler generated code that would replicate one of the arrays to each processor every time the gather was executed. In contrast, shared memory referencing relaxes the compiler rules and does not cause this replication. Rather, the compiler generates in-line calls to optimized Cray communications libraries. Very limited testing was required to determine that the cost of replication would well exceed any benefit from generic block use. There is no limiting reason that precludes the combination of shared memory referencing with asymmetric block distribution. Hopefully this will be rectified in future compiler releases.

4. Performance Data

For two reasons, the discussion of the performance of HPF is limited to the Cray T3E system alone. First, the ability to reuse communication schedules is only supported on the HPF compiler for the Cray T3E. Second, PGHPF has been highly optimized for the Cray T3E system compared to other major architectures [11]. The wall clock execution times for a representative problem with varying processor counts are given in Table 1. This table lists the time of execution using reusable communication schedules for the `SUM_SCATTER` operation. Since we overwhelmingly showed the need for reusable schedules in the 16-processor case, we considered further benchmarking using the `SUM_SCATTER` library routine to be a waste of CPU hours. Table 1 shows the sometimes dramatic improvement of simple mesh preprocessing for data parallel execution.

Table 1: Execution Times (Seconds) for a Mesh Analysis on a Cray T3E.

Problem		Number of processors				
		2	4	8	16	32
Airframe structure	Original Mesh	20951.89	13117.40	8515.41	6111.37	3603.69
	Renumbered Mesh	19189.21	9989.31	5553.45	3135.03	2095.47

The limits to speedup appear to reside wholly in areas of the code requiring interprocessor communication. As such, the ability to perform precommunication reductions and reuse communication schedules appear to be the best opportunity at increasing code speed. The hope is that asymmetric block distribution and schedule reuse be incorporated into future releases of the compiler. Mesh renumbering, combined with these techniques, present the best hope for making HPF viable for unstructured grid problems.

Currently, MPI provides the best wall clock solution for this class of problems. More information will be provided in future monographs, but an example is telling. On one comparision problem (45,547 nodes, 89,945 elements), the MPI implementation of COMPOSE

required about 1038 s to complete on a 16-processor run on the Cray T3E system. The preprocessed and optimized HPF run required almost 3× as long at 2949 s. HPF must overcome its current limitations to be more broadly applicable.

5. Conclusion

The HPF language and the data parallel model it is built upon have both shown applicability to past, current, and no doubt future parallel computer platforms. As more and more choices become available to achieve parallelism, it should be pointed out that there is no single panacea for parallel programming. Some researchers believe that data parallel is a natural way of thinking and achieving parallelism, compared to something like shared memory programming or message passing. Others, first exposed to parallelism with loop-level directives or MPI, would probably say that shared memory or SPMD programming is more natural. To a large extent, it no doubt depends on the individual and past history.

Furthermore, it is also difficult to judge effort and payoff between various parallel programming tools and languages. Some researchers have expressed beliefs that it is easier to write and maintain code in HPF rather than message passing with MPI [12]. Such statements are misleading. Good performance, however measured, always takes time to achieve. HPF code requires careful and time consuming tuning, as does a code written using other languages and libraries.

In summary, HPF provides a viable parallel programming capability for some classes of problems, but it needs improvements. The standard suffers by not addressing very important considerations like interprocessor communication schedules. These issues can only be addressed at the compiler and standardization levels. No amount of code tuning can correct these problems. Therefore, it is up to the individual to decide if this degraded performance is acceptable over some possibly more "expensive" alternative. As the compilers and the standard continue to mature, they will only improve—their survival depends upon it.

6. References

[1] Chapman, S. J. *Introduction to Fortran 90/95*. Boston, MA: McGraw-Hill, 1998.

[2] Ellis, T. M., I. R. Philips, and T. M. Lahey. *Fortran 90 Programming*. Reading, MA: Addison-Wesley, 1994.

[3] Koelbel, C. H., D. B. Loveman, R. S. Schreiber, G. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. Cambridge, MA: MIT Press, 1994.

[4] Kumar, V., A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Reading, MA: Benjamin/Cummings, 1994.

[5] Boucher, M. "HPC Performance and Optimization: Libraries, Techniques, and Tools for Sun's HPC Product Family." Lecture at the U.S. Army Research Laboratory Major Shared Resource Center (MSRC), 1999.

[6] Pfister, G. F. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing*. Upper Saddle River, NJ: Prentice Hall, 1995.

[7] Foster, I. *Designing and Building Parallel Programs*. Reading, MA: Addison-Wesley, 1995.

[8] Chilimbi, T., M. Hill, and J. Larus. "Making Pointer-Based Data Structures Cache Conscious." *Computer*, vol. 33, no. 12, pp. 67–74, 2000.

[9] Portland Group. Private Communication with Director of Marketing, 1999.

[10] Cuthill, E. and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices." In *24th National Conference*, Association for Computing Machinery, pp. 157–172, 1969.

[11] Miles, D. Portland Group. Presentation at the U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, June 2000.

[12] Luecke, G. and J. Coyle. "High Performance Fortran Versus Explicit Message Passing on the IBM SP-2 for the Parallel LU, QR, and Cholesky Factorizations." Technical Report, Iowa State University, 1997.

NO. OF COPIES	ORGANIZATION
2	DEFENSE TECHNICAL INFORMATION CENTER DTIC DDA 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	HQDA DAMO FDT 400 ARMY PENTAGON WASHINGTON DC 20310-0460
1	OSD OUSD(A&T)/ODDDR&E(R) R J TREW THE PENTAGON WASHINGTON DC 20301-7100
1	DPTY CG FOR RDA US ARMY MATERIEL CMD AMCRDA 5001 EISENHOWER AVE ALEXANDRIA VA 22333-0001
1	INST FOR ADVNCD TCHNLGY THE UNIV OF TEXAS AT AUSTIN PO BOX 202797 AUSTIN TX 78720-2797
1	DARPA B KASPAR 3701 N FAIRFAX DR ARLINGTON VA 22203-1714
1	US MILITARY ACADEMY MATH SCI CTR OF EXCELLENCE MADN MATH MAJ HUBER THAYER HALL WEST POINT NY 10996-1786
1	DIRECTOR US ARMY RESEARCH LAB AMSRL D D R SMITH 2800 POWDER MILL RD ADELPHI MD 20783-1197

NO. OF COPIES	ORGANIZATION
1	DIRECTOR US ARMY RESEARCH LAB AMSRL DD 2800 POWDER MILL RD ADELPHI MD 20783-1197
1	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AI R (RECORDS MGMT) 2800 POWDER MILL RD ADELPHI MD 20783-1145
3	DIRECTOR US ARMY RESEARCH LAB AMSRL CI LL 2800 POWDER MILL RD ADELPHI MD 20783-1145
1	DIRECTOR US ARMY RESEARCH LAB AMSRL CI AP 2800 POWDER MILL RD ADELPHI MD 20783-1197
	ABERDEEN PROVING GROUND
4	DIR USARL AMSRL CI LP (BLDG 305)

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2001	Final, June 2000 - January 2001

4. TITLE AND SUBTITLE
Strategies and Experiences Using High Performance Fortran

5. FUNDING NUMBERS
665803.731

6. AUTHOR(S)
Dale Shires, Ram Mohan, and Andrew Mark

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-HA
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-2437

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITYSTATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT*(Maximum 200 words)*

Since its beginnings in the 1950's, the Fortran language has been the language of choice for most scientific and engineering programming. Compilers, seizing on the simplicity of the language, have historically generated highly-optimized machine code. High performance Fortran (HPF) is a relative new addition to the Fortran dialect. It is an attempt to provide an efficient high-level Fortran parallel programming language for the latest generation of parallel computers. Its success has been debatable. By operating at a high level, the HPF standard does not provide some low-level detail required to achieve maximum performance in a multiprocessor system. Message passing using highly-tuned libraries, such as the message passing interface (MPI), will more often than not require less wall clock time than a comparable HPF code. However, the HPF language and its compilers continue to mature and improve. HPF provides a convenient way to represent parallelism for those most comfortable with data parallel methodologies. As such, it can require a shorter time to solution and provide an acceptable level of efficiency. This report discusses our experiences with the language, as well as coding strategies and vendor-specific "hooks" that can be used to boost performance.

14. SUBJECT TERMS
data parallelism, high performance Fortran, HPF, code optimization

15. NUMBER OF PAGES
35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number/Author ARL-TR-2437 (Shires) Date of Report March 2001

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

_____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

_____

_____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

_____

_____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

_____

_____

_____

CURRENT ADDRESS

_____
Organization

_____
Name E-mail Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)

(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001,APG,MD

POSTAGE WILL BE PAID BY ADDRESSEE

DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL CI HA
ABERDEEN PROVING GROUND MD 21005-5067